Palo Alto Research Centers

An Analysis of a Mesa Instruction Set

Gene McDaniel

XEROX

An Analysis of a Mesa Instruction Set

Gene McDaniel

CSL-82-2 **May 1982** [P82-00031]

Abstract: This paper reports measurements of dynamic instruction frequencies for two Mesa programs running on a Dorado personal computer at the Computer Science Laboratory of the Xerox Palo Alto Research Centers. The patterns of use associated with the Mesa instruction set are examined in order to find the implications of that usage for the Mesa architecture and its implementation. This paper discusses Mesa's byte encoding, patterns of memory references, use of an expression evaluation stack, and the costs of emulating 32-bit operations on a 16-bit processor.

A version of this paper appeared in *The Proceedings of the Symposium on Programming Languages and Operating Systems*, December 1981, Palo Alto California.

CR categories: B.1.5, C.4, D.3.4, D.4.8.

Key words and phrases: architecture, instruction set analysis, microcode emulation, performance analysis.

XEROX

Xerox Corporation
Palo Alto Research Centers
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

This paper reports measurements of dynamic instruction frequencies for two Mesa [1,2] programs running on a Dorado personal computer [5] at the Computer Science Laboratory of the Xerox Palo Alto Research Centers. The purpose of this study is to examine the patterns of use associated with the Mesa instruction set, and to find the implications of that usage for the Mesa architecture and its implementation. This paper discusses Mesa's byte encoding, patterns of memory references, use of an expression evaluation stack, and the costs of emulating 32-bit operations on a 16-bit processor.

Mesa is a high level, systems implementation language with a strong, flexible type machinery that permits independently compiled programs to be combined as a functional unit. The Mesa compiler generates instructions that run on any processor that implements the Mesa architecture. The Mesa architecture defines the instruction set and other run time facilities necessary to support Mesa programs. There are a variety of implementations of Mesa, including one for the Dorado. The work described here was done with a prototype version of the Mesa architecture. The differences between the two architectures are not consequential to this study. The official architecture is described elsewhere [8].

2. Experimental Method

The author modified the Dorado's microcoded, Mesa emulator to keep statistics on instruction frequencies. The microcode maintained an array of 2^{16} 32-bit counters, where each element in the array contained the count of the number of times a particular *pair* of instructions executed. The microcode was modified to save the opcode byte of the last instruction, and that byte was concatenated with the current opcode byte to provide an index into the array of instruction pair counts. Single instruction frequencies were computed from this data. This bookkeeping activity slowed the Mesa emulator by about a factor of six.

The Mesa run time system was modified so that this counting facility was disabled while the system executed disk-wait loops. The measurements did not distinguish between time spent in the operating system and time spent in the two programs described below.

While different instructions require different amounts of time to execute, only instruction frequency information was collected. This paper does not make a study of the execution times (duration) for the various instructions (such information is crucial to the analysis of any implementation of the Mesa virtual machine).

The two programs examined were the Mesa compiler and a VLSI circuit analysis program, VlsiCheck. The compiler was designed to execute on an Alto [4] with only 64K words of 16-bit memory. It comprises about 39,000 lines of source, and is an example of a program highly optimized for space efficiency. The compiler is a production program, and to enhance its speed, it does not use bounds or pointer checking. In contrast, VlsiCheck, which is about 500 lines of source, deals with 32-bit quantities (both data and pointers), and may require 1M word of memory or more. VlsiCheck is a relatively new program that makes full use of bounds and pointer checking.

3. Mesa's Run Time Structures

Mesa is a high level, systems implementation language that provides for facilities such as a strong type system and separate compilation of modules. While the ability to declare new data types at compile time is a crucial part of the language, it is not interesting from an architectural perspective, since the Mesa instruction set provides no particular accommodations to the type machinery. See [1,2] for more information about the language, [8] for a more detailed discussion of the run time system, and [9] for details about the byte encoding.

Procedures and Modules. In a Mesa program, code executes as part of a procedure body or module body. All procedures exist in the context of a module, and a procedure's data exists only for the duration of the procedure's execution. A module, however, contains information with a longer lifetime, i.e., as long as the module exists in an executing Mesa program. The Mesa processor architecture supports two run time structures that parallel this arrangement: for each instance of a module there is a unique structure known as the Global Frame that contains module specific information, and for each activation of a procedure there is a unique structure known as the Local Frame that contains procedure specific values.

Memory Access. Memory that is addressed by 16-bit pointers lives in an area known as the Main Data Space, which contains the Local and Global frames as well as some structures allocated by the programmer. The Mesa architecture defines dedicated registers to hold pointers to the current Local (L) and Global (G) frames, and the Main Data Space (MDS). Much of the Mesa instruction set is concerned with reading or writing data relative to one of these three pointers. Access to memory outside the MDS is described below.

Byte Encoded Instruction Set. The Mesa instruction set employs a compact form of byte encoding [9,10,11] wherein the first byte is the opcode of the instruction and subsequent bytes are operands for the instruction. The instruction set has been optimized to provide a high degree of code compaction: the instructions that occur most frequently (by static measurements) are one-byte long, those that occur less frequently two-bytes long, etc. [9]. Code compaction supports the goal of being able to make large, complex systems that can run efficiently in machines with relatively small amounts of primary memory [8].

The Mesa compiler facilitates code compaction in an important way: variables in the Local and Global frames are allocated in an order that reflects their static frequency of use. The compiler sorts variables by decreasing order of static occurrences (the number of occurrences in the program text), and places them in that order in the Local or Global frames. Since the most frequently occurring instructions are encoded in one-byte, the most frequently occurring variables can be referenced with one-byte instructions.

In this paper, the terms $Local_i$ and $Global_i$ refer to the i^{th} entries in the respective frame. For example, Load Local 4 (LL4) is a one-byte instruction, but Store Global 23 (SGB 23) uses two-bytes where the second byte provides an eight-bit offset.

Expression Evaluation Stack. The Mesa architecture defines a small expression evaluation stack that can be implemented in the high speed registers of the processor. The architecture defines the minimum size of the stack, and all ALU operations (arithmetic, Boolean) take operands from the stack and leave results on the stack. Consequently, neither the ALU instructions nor the load and store instructions, which move items between memory and the stack, require bits to specify machine register addresses.

4. The Mesa Instruction Set

This section provides a brief overview of the Mesa instruction set. The instruction set provides a collection of general purpose operations and some special purpose ones.

Memory References. Memory instructions either operate relative to one of the pointers discussed in §3 (L, G, MDS), or provide a full (32-bit) address. In addition to instructions that deal with 16-bit quantities, there are instructions that support the use of 32-bit quantities such as LONG POINTER, LONG INTEGER, etc.

The compiler decides the locations of variables within the Local and Global frames, and the Mesa run time-system allocates the storage for those frames at run time. The programmer may allocate storage for additional structures within or outside the MDS. There are instructions that perform miscellaneous operations on memory data such as reading or writing bytes from a string, moving a block of bytes, Raster Op [4], etc.

Records. The compiler may use one or more instructions to access the components of a record. To support records, there are special instructions to load or deposit contiguous bit fields within a 16-bit value.

Branches and Control Transfers. The normal flow of program execution through a sequence of consecutive code bytes may be interrupted because of branches or control transfers. A branch is a conditional or an unconditional jump, and a variety of them are defined. The opcode byte defines the type of branch and any data bytes provide the PC-relative offset. Control transfers are exemplified by procedure call and return, coroutine transfers, and process switches. The instruction set likewise defines a variety of instructions to support these activities.

Arithmetic and Boolean Operations. There are a collection of arithmetic and Boolean instructions that operate on the top two elements of the evaluation stack and leave their results on the stack. While these instructions default to 16-bit quantities, some work on 32-bit quantities. Floating point instructions are implemented with traps if there is no supporting microcode or hardware.

5. Partitioning the Instruction Set into Different Groups

To avoid the blizzard of detail associated with data collected for this study, much of this paper discusses the Mesa instruction set in terms of different groups of instructions. A collection of groups that contains the entire instruction set is a partition, and five partitions of the instruction set are described below. Each partition provides a different insight into the use of the Mesa instruction set.

The Stardard Partition. This partition reflects a division of the instruction set into ten groups that are most obviously defined by the architecture, and is very similar to one documented elsewhere [8].

- Load/Store: Load or Store values between the evaluation stack and memory locations relative to the Local or Global frames.
- Load Immediate: Load constant values onto the evaluation stack from the code stream.
- R/W: Read or write values relative to a pointer. The pointer may be MDS relative or a "LONG" pointer.

- ALU Ops: Perform an arithmetic or Boolean operation and push the results onto the top of the evaluation stack.
- Stack Ops: Change the stack pointer only (this may recover "popped" data).
- Jumps: These are unconditional jumps; they always execute.
- Conditional Jumps: These are local jumps that happen only if the condition specified by the opcode is true.
- Xfers: These are non-local transfers [3], such as procedure calls and returns (as opposed to conditional and unconditional jumps).
- Process: These instructions identify process operations.
- Miscellaneous: Various instructions that do not fit into any other categories.

Locals vs. Globals. This partition distinguishes those instructions that reference data through the Local or Global frames from all other instructions. This includes instructions like "Load Local 3" and indirect instructions that reference data through pointers in frames. However, not all instructions that *might* touch the Local or Global frames are included in this group, since instructions that deal with pointers, procedure call and return, etc., are not included.

- Locals: Move values between the expression evaluation stack and the Local frame.
- Locals Indirect: Move values between the expression evaluation stack and memory through a pointer in the Local frame.
- Globals: Similar to "Locals", except the references are to the Global frame.
- Globals Indirect: Similar to "Locals Indirect", except the references are through the Global frame.
- All other instructions.

Loads vs. Stores. This partition distinguishes instructions that load values onto the stack from memory from the ones that store values from the stack into memory.

- Loads: These instructions read values from memory and place them in the evaluation stack.
- Stores: These instructions write values from the evaluation stack into memory.
- All other instructions.

Components for Memory Address. This partition distinguishes the number of components that must be added together to compute the effective address for an instruction that references memory. Instructions that use multi-component addresses must use an ALU to compute the effective address of the instruction. For simplicity of analysis we distinguish only two of the many possible cases, and instructions that deal with more than one data item are in the category of "other".

- Mem1: These instructions use a single-component address.
- Mem2: These instructions use a two-component address.
- All other instructions.

Instruction Length. This partition distinguishes instructions based upon their instruction length.

- Length1: These instructions are only one-byte long.
- Length2: These instructions are two-bytes long.
- Length3: These instructions are three-bytes long.

6. Single Instruction Frequencies

A Short Glimpse at Individual Instructions. It is hard to learn much of *general* interest by examining individual instruction frequencies of a small set of programs, since those statistics may highlight idiosyncratic behavior of the programs. For example, only three of the top eight most frequently executed instructions in both VlsiCheck and the compiler are the same. (See the table below).

The compiler's favorite instruction is a conditional branch while VlsiCheck's is a double-word load. The compiler concerns itself with making discriminations about the information it has in hand, while VlsiCheck continually searches a large data-base outside the MDS. This means that VlsiCheck must make extensive use of 32-bit pointers and data.

A Few of the Most Frequent Instructions

Complier	%	Sum	VlsiCheck	%	Sum
Jump If ≠0	10.30	10.3	Load Local Double-Word	7.04	7.04
Load L0	8.96	19.26	Load L0	6.39	13.43
Read Field	7.50	26.76	Store Local Double-Word	5.15	18.58
Load Immed 16-bit Value	5.51	32.27	Recover Stack Item	4.93	23.51
Add	4.94	37.21	Load Immed Byte (8-bit value)	4.60	28.11
Read Indirect	4.6	41.81	Load Immed. 0	3.92	32.03
Recover Stack Item	3.51	45.32	Read Indirect Index Off Pointer	3.11	35.14
Load G0	2.99	48.31	Jump If ≠0	3.03	38.17

Single Instruction Frequencies by Partition: The Standard Partition. Recall from §5 the groups of the Standard Partition reflect the designer's view of the instruction set: ALU operations, loads and stores, jumps, etc. The two tables appearing on the following page show the relative frequencies for the standard groups. In summary, they show that most instructions move data in and out of memory, there are a moderate number of transfers, and there are relatively few ALU operations.

The most important result shown by the first table is that for both programs, about 50% of all instructions are primarily concerned with reading or writing memory. The Xfer and Process groups also reference memory, though their functions are different. Thus, the speed of memory references is an important parameter in any Mesa processor implementation.

The second table shows statistics about branches, conditional branches, and procedure calls. These are interesting because of the effect they have upon instruction prefetch hardware in a computer–they interrupt the expected, sequential processing of the code stream [13].

Statistics For "Standard" Partition

Compiler			VlsiCheck		
Group	%	Sum	Group	%	Sum
Ld/Store	32.97	32.97	Ld/Store	35.15	35.15
R/W	19.59	52.57	R/W	14.14	49.29
CondJumps	16.82	69.39	Stack Ops	12.23	61.52
Ld Immed	11.43	80.82	ALU Ops	10.76	72.28
ALU Ops	8.14	88.96	Ld Immed	10.53	82.81
Stack Ops	3.87	92.84	CondJumps	8.42	91.23
Xfers	3.55	96.39	Xfers	5.31	96.54
Jumps	2.25	98.64	Jumps	1.75	98.29
Misc	1.35	99.99	Misc	1.67	99.96
Processes	0.01	100.0	Processes	0.04	100.0

Branches, Xfers, and Jumps

Compiler			VlsiCheck		
Type	%	Sum	Type	%	Sum
CondJumps	16.82	16.82	CondJumps	8.42	8.42
Xfers	3.55	20.37	Xfers 5.31	13.73	
Jumps	2.25	22.62	Jumps	1.75	15.48

The tables and figures below show the most frequently executed instructions within each group of the Standard Partition. For the sake of brevity, only the first three or four instructions in each group are shown. Note that within each group only a few instructions account for most of the activity in that group, and that bounds and NIL checking (in Stack Ops group) cost only 5.14% of all instructions, even in a program like VlsiCheck, that extensively reads and writes memory.

Opcode mnemonics are provided in the appendix.

Compiler Percentages				VlsiCheck Percentages			
Instr	Group	Over All	Sum	Instr	Group	Over All	Sum
Ld/Store=32.97% Over All				Ld/Store=35.15% Over All			
LL0	27.16	8.96	8.96	LLDB	20.04	7.04	7.04
LG0	9.06	2.99	11.95	LL0	18.17	6.39	13.43
LL1	7.29	2.40	14.35	SLDB	14.65	5.15	18.58
LL2	5.02	1.76	20.34				
R/W=19.59% Over All				R/W=14.14% Over All			
RF	38.23	7.49	7.49	RILP	21.96	3.11	3.11
R0	23.68	4.64	12.13	R0	13.00	1.84	4.95
RXLP	6.86	1.34	13.47	RDBL	10.38	1.47	6.42
				RSTR	9.66	1.37	7.79
CondJumps=16.82% Over All				Stack Ops=12.23% Over All			
JZNEB	61.18	10.29	10.29	PUSH	40.30	4.93	4.93
JZEQB	7.87	1.32	11.61	NILCKL	21.78	2.66	7.59
JEQB	5.71	.96	12.57	BNDCK	10.37	1.33	8.92
				NILCK	9.42	1.15	10.07
Ld Immed=11.43% Over All				ALU Ops=10.76% Over All			
LIW	47.31	5.41	5.41	MUL	24.16	2.60	2.60
LIB	13.95	1.59	7.00	ADD	21.62	2.33	4.93
LI0	13.49	1.54	8.54	SUB	12.98	1.40	6.33
LI1	9.79	1.12	9.66	INC	11.33	1.22	7.55

Reading the Graphs. Solid lines represent the cumulative contribution of instructions to their group, and the dashed lines represent their cumulative contribution to all instructions executed. Note, the histograms are *not* sorted by frequency in order to make differences between instruction frequencies easier to notice.

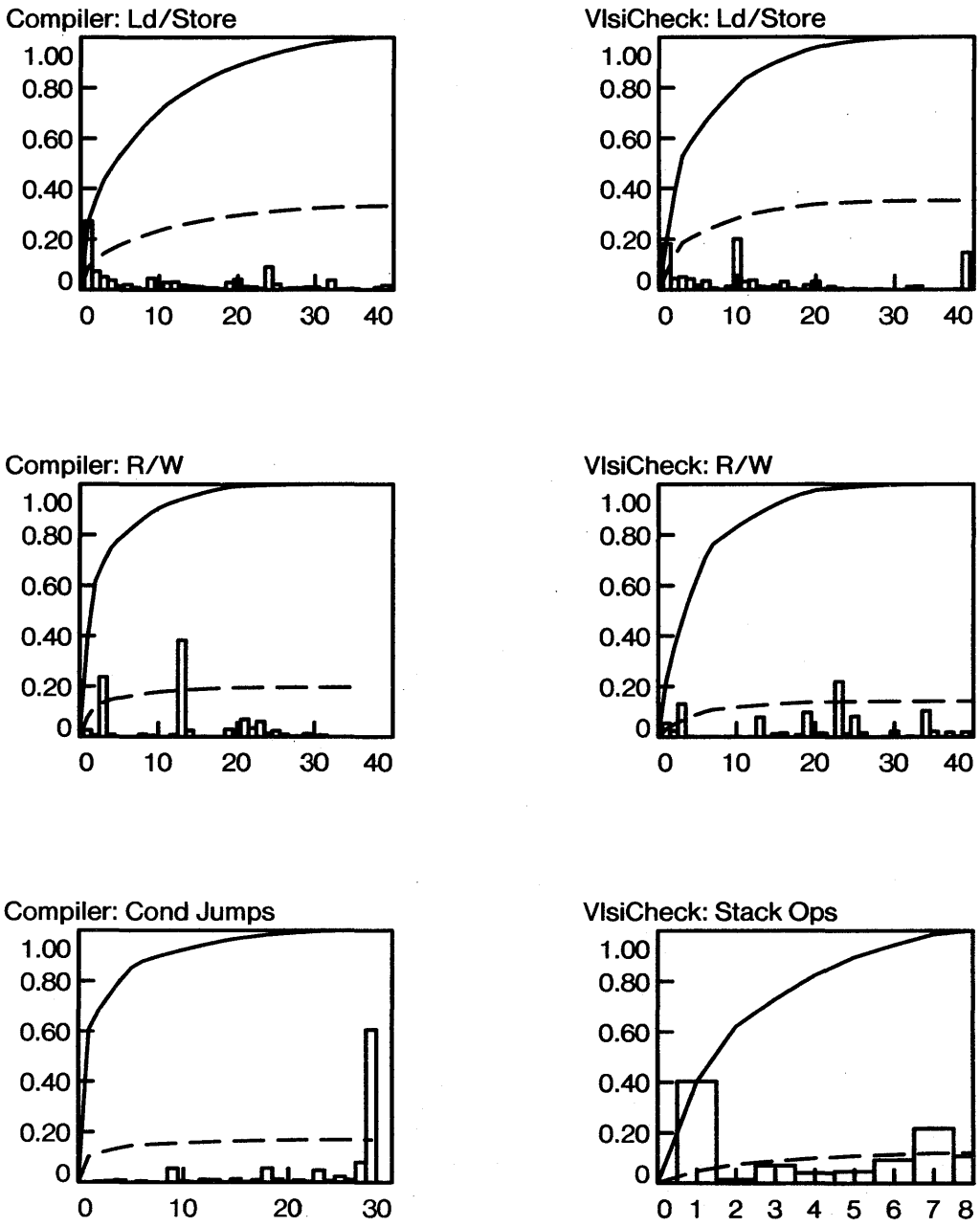

FIGURE 1.

Single Instruction Frequencies by Partition: Locals vs. Globals. This group compares instructions that reference data in the Local or Global frames. The code of a procedure can easily and directly access both its Local and Global variables. All other data must be accessed through pointers. The table and Figure 2 provide details about the data.

Statistics For "Locals vs. Globals" Partition

Compiler			VlsiCheck		
Group	%	Sum	Group	%	Sum
Locals	26.44	26.44	Locals	34.29	34.29
Globals	7.20	33.64	Globals	1.95	40.96
LocalsIndirect	3.84	37.48	LocalsIndirect	4.72	39.01
GlobalsIndirect	0.28	37.76	GlobalsIndirect	0.17	41.13

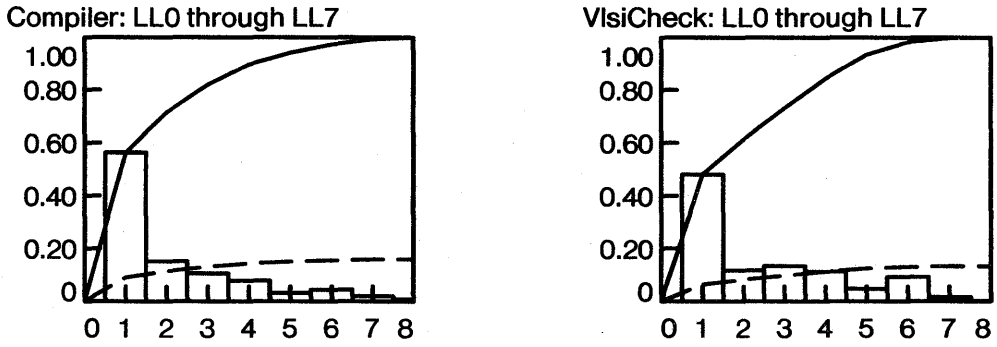

FIGURE 2.

The Locals and Globals groups account for most of the references – explicit indirection through those frames was relatively infrequent (except VlsiCheck's LocalsIndirect). The total percentage of all references into those frames is greater since some pointer instructions may reference Local or Global data. By far, the preponderance of references went to the local frame: 30.28% (compiler), 39.01% (VlsiCheck). In other words, 70%-80% of "Local or Global frame instructions" referenced the Local frame. These statistics support intuition which expects local storage to be very heavily used.

Notice that VlsiCheck uses more load double-word instructions than load local zero instructions. This follows from two facts: First, VlsiCheck deals with many double-word quantities, and second, there is no single-byte instruction that loads double-word quantities beginning at Local frame 0 or Local frame 1, etc. The load double-word instruction is a two-byte instruction, in which the second-byte specifies the offset into the Local frame, wherein the double-word quantity begins. The compiler will allocate a double-word quantity first in the frame if the static references justify doing this. Since there is no "Load Local 0 Long" instruction, the references to Local 0 decrease.

Single Instruction Frequencies by Partition: Loads vs. Stores. This partition distinguishes memory read and write instructions from all others. There is considerable difference between the ratio of loads to stores in the two programs: for VlsiCheck, it is 2:1 loads to stores, and for the compiler it is almost 4:1. Of course, instruction fetch references, which are not included here, will increase the statistics in favor of loads. Such references may not be performed by the processor. The table on the following page compares the statistics for the two programs.

Statistics For "Loads vs. Stores" Partition

Compiler			VlsiCheck		
Group	%	Sum	Group	%	Sum
Loads	40.64	40.64	Loads	32.79	32.79
Stores	11.27	51.91	Stores	15.40	48.19
Others	48.09	100.0	Others	51.81	100.0

Figure 3 shows that the distribution of opcode frequency usage for Loads in the compiler is more skewed than it is for Stores: 8 load instructions account for more than 76% of the Loads group, while 12 store instructions are needed to approach 75%. VlsiCheck, on the other hand, shows approximately the same distribution for both load and store instructions.

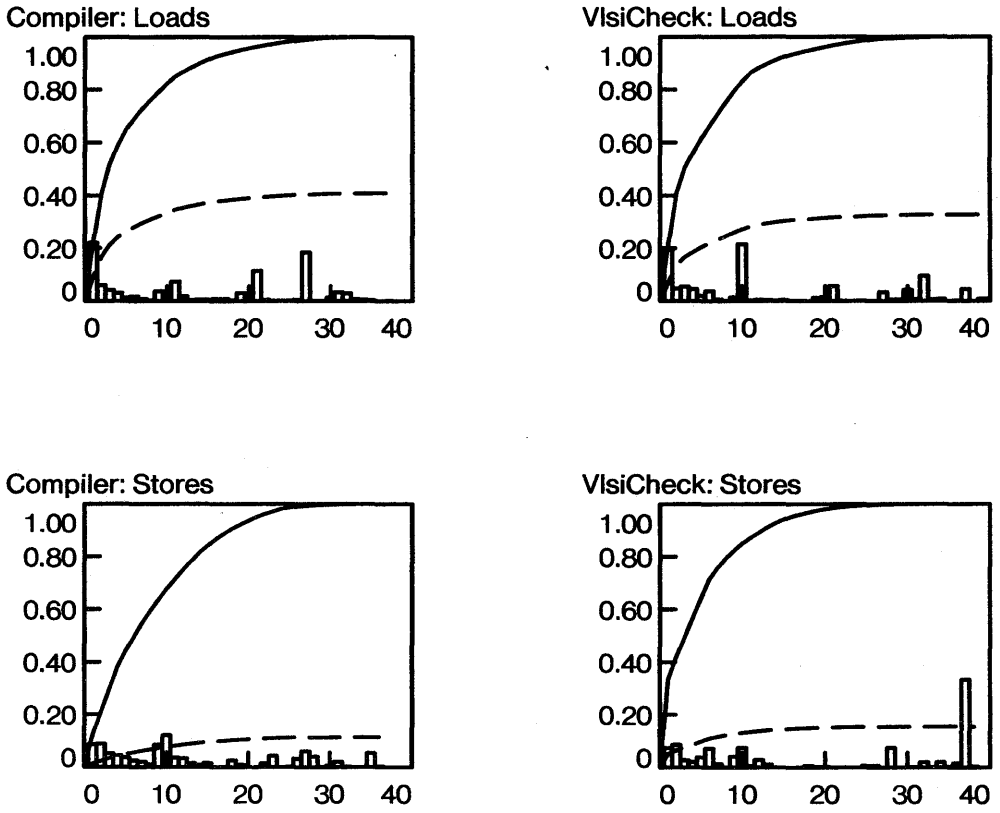

FIGURE 3.

Single Instruction Frequencies by Partition: Memory Address Components. This partition distinguishes memory reference instructions on the basis of whether an ALU is required to compute the virtual address. Members of the Mem1 group do not require an ALU operation – they could be performed by an operand prefetch unit in the processor. The table appearing on the next page shows the pattern of use for instructions in this group. For the compiler, over 19% of all instructions, almost 1 in 5, are of type Mem1 where the offset is zero, while for VlsiCheck this value decreases to approximately 11%.

The Mem2 group accounts for about 17% of the compiler's and 25% of VlsiCheck's instructions. Remember that this group includes the instructions that load or store, given a pointer and a single offset; it does not include complicated instructions that perform more than one memory reference, even though the first reference is of type Mem2. The remaining memory references executed by those programs are distributed across instructions with complicated operands for the memory operation.

Statistics For "Memory Components" Partition

Compiler			VlsiCheck		
Group	%	Sum	Group	%	Sum
Mem1	19.18	19.16	Mem1	10.94	10.94
Mem2	16.78	35.94	Mem2	24.81	35.75

Single Instruction Frequencies by Partition: Instruction Length. Dynamic instruction length frequencies indicate how much bandwidth there must be between the memory and the processor to keep the processor busy with instructions. The advantage of Mesa's compact instruction set becomes evident here, since both programs executed one-byte instructions more than half the time, and the average instruction length is about 1.5 bytes.

Statistics For "Instruction Length" Partition

Compiler			VlsiCheck		
Group	%	Sum	Group	%	Sum
Length1	55.22	55.22	Length1	56.72	56.72
Length2	38.64	93.86	Length2	41.66	98.38
Length3	6.14	100.0	Length3	1.62	100.0
Average Length	1.51		1.45		

7. Pairwise Instruction Frequencies

Pairwise instruction data presents a large quantity of detail, and both programs show idiosyncracies at this level of detail. For the compiler, the 140 most frequent pairs account for only 30% of all pairs executed, and for VlsiCheck, they account for only 37% of all pairs. The data was analyzed by grouping instructions as members of various groups, and by attempting to infer information about the character of usage based upon group membership.

For example, the pairwise data associated with the compiler shows that load immediate instructions were used frequently as a parameter for conditional branches. While the processor executes the load immediate instruction, it does not utilize many of the resources available to it. In particular, if there is an instruction prefetch unit that has decoded the instruction and provided its parameters to the processor, the processor itself is not using the memory system during the time it pushes the constant onto the evaluation stack. During this interval the processor's memory port is idle, and thus available for some other use. (See the following table). Sweet and Johnsson [9] note a similarity in the static statistics, and they recommend new, conditional branch instructions that provide small constants from the alpha byte of the conditional branch.

Statistics For "Any Group, Condition Jump" Pairs

Compiler			VlsiCheck		
Group	%	Sum	Group	%	Sum
Memory Loads	12.90	12.90	Memory Loads	4.23	4.23
Ld Immed	3.07	15.97	Ld Immed	2.20	6.43

Memory Loads and Stores. Consider the set of pairs, "memory load or store" followed by any other group. Of all the times the compiler executes a memory load instruction, about a third of the following instructions is a branch, about a third of them is another load instruction, and about 15% of the following instructions are stack or ALU operations.

Conditional Branches. The frequency and behavior of conditional branches may have considerable impact on a machine due to the necessity of acquiring the target instruction from the memory. For example, the IBM 370/91 [12] and 3033 [15] have hardware to prefetch instructions down both branches of a conditional jump. Simpler machines may implement instruction fetching using a microcoded emulator with no specific hardware support.

The compiler's conditional branches are preceded by instructions that load the processor with data and they are followed with instructions of the same sort. When the data being loaded comes from memory, there is a contention between the need for an instruction reference and the need for a data reference to memory (unless the code and data are kept in separate memories that can be accessed simultaneously). See the table below.

Statistics For "Any Group, Condition Jump" Pairs

Compiler			VlsiCheck		
Group	%	Sum	Group	%	Sum
R/W	10.37	10.37	R/W	2.92	2.92
Ld Immed	3.07	13.44	Ld Immed	2.19	5.11
Ld/Store	2.93	16.37	Ld/Store	1.78	6.89
Stack Ops	0.25	16.62	ALU Ops	1.74	8.63

Statistics For "Condition Jump, Any Group" Pairs

Compiler			VlsiCheck		
Group	%	Sum	Group	%	Sum
Ld/Store	9.09	9.09	Ld/Store	4.31	4.31
Ld Immed	4.89	13.98	Ld Immed	1.68	5.99
Stack Ops	1.81	15.79	Stack Ops	0.83	6.82
R/W	0.29	16.08	ALU Ops	0.47	7.29

The most frequent instruction that precedes a conditional jump (for the compiler) is the read field instruction – 40.86% of the pair "load some value from memory, conditional jump." This is 5.27% of all the pairs executed by the compiler, and probably reflects idiosyncratic behavior.

Bounds and NIL Checking. The importance of stack operations to VlsiCheck is an anomaly due to classification: the instructions that perform bounds and NIL checking are classed here as stack operations, and they account for about 5.1% of all instructions. These instructions precede instructions of the Ld/Store and R/W groups. If 5.1% is subtracted from the percentage for stack operations in VlsiCheck (12.23%), the remainder, which indicates the per cent of "pure" stack instructions, is about 7% – nearly twice the figure for the compiler. The factor of two probably stems from executing the "recover" operation on the stack twice as often because of 32-bit values in the stack: The recover instruction regains old values in the stack by incrementing the stack pointer without pushing a value. This trick acquires old values without making memory references. Of course, the compiler's code generator is responsible for assuring the recovered value is still valid.

The NIL checking instructions raise another issue. If the processor has hardware support for a virtual memory, most of the NIL checking instructions can be removed by making NIL a distinguished value that is not mapped. Processor references to it would cause a page fault and that could be translated into an attempt to dereference NIL by the software. This could save VlsiCheck up to 3.8% of its instructions. However, pointers that are indexed may still require explicit checking since the value of "NIL+index" may be a legitimate, mapped address.

Load Immediate Instructions. These instructions exist because of the way the Mesa instruction set has been optimized to minimize code size. In many other architectures, ALU and memory instructions permit relatively large constants as parameters, whereas many Mesa instructions acquire constant parameters from the stack where preceding instructions have left them. The most frequent uses for load immediate instructions is to provide a parameter for a memory reference, for a conditional branch, or for an ALU operation.

Some of the statistics associated with the set of pairs that follow load immediate instructions are shown below.

Statistics For "Load Immediate, Some Group" Pairs

Compiler			VlsiCheck		
Group	%	Sum	Group	%	Sum
R/W	5.11	5.11	ALU Ops	2.60	2.60
Cond Jump	3.07	8.18	Cond Jump	2.20	4.80
ALU Ops	1.53	9.71	R/W	1.34	6.14

8. Discussion

Compiler's Variable Location Optimizations. Recall from §3 that the compiler chooses the locations of variables in a frame based upon their static frequency of usage. This provides for superior code compaction, since one-byte instructions thereby reference most variables. The statistics presented show that the compiler's static analysis of variable usage also successfully predicts dynamic usage. Local and Global zero are the most frequently used Local and Global variables. Figure 2 shows the frequency of variable references of Locals 0-7. Notice that the frequencies do not monotonically decrease, and that the aberrations represent a very small fraction of all instructions. These variations are due, of course, to the compiler's inability to predict perfectly which variables will be accessed most frequently at run time.

Caching Local Variables. If Locals and Globals 0-3 are kept in a register cache, up to 23.98% of the compiler's instructions, and 16.63% of VlsiCheck's instructions can be transformed from memory-reference instructions into register-to-register instructions. The advantage of this depends upon the speed difference between memory and the registers. This arrangement requires care since it must work properly in the presence of pointers, procedure calls, and process switches. Lampson [14] discusses the advantages of this approach to implementing speedy procedure calls.

Memory Components. Ten to twenty percent of the instructions were full pointer (Mem1) instructions which do not require an ALU operation in the processor to compute their virtual address. This suggests that fairly simple operand prefetch hardware might provide substantial performance improvement in a Mesa processor. The problem with operand prefetching is the hardware interlocks that must be implemented in order to assure that an address used by the prefetch hardware is not being modified by the processor. Once such interlocks are present, it might be worthwhile to prefetch Mem2 type operands as well.

A different advantage might be obtained by speeding up memory references that use full pointers, instead of trying to speed up those that require the ALU for arithmetic on two or more operands before the effective address has been computed.

The use of base registers in the memory system that point to the base of the Local and Global frames [5] would change all the direct Local and Global memory instructions (the Ld/Store group) into full pointer instructions, thereby eliminating the need for an ALU in the processor for over 30% of all instructions that reference memory.

Presumably, most of the advantages associated with the Meml instructions are directly due to the compiler's choice of Local 0 and Global 0.

The Implications of the Stack. Typical machine architectures define a processor with a set of general purpose registers instead of an expression evaluation stack. The chief advantage of a register architecture is to provide a cache for frequently used values. The disadvantages are the extra costs associated with saving and restoring the registers across procedure calls and context switches, and decreased code compaction because register addresses must be provided within most instructions. The chief advantage of a stack oriented architecture is the code compaction: Johnsson and Wick provide an example program that shows a reduction from 36 bytes to 7 bytes by using a stack architecture, although more typical results show a factor of two compaction [8]. The disadvantage of an evaluation stack is the lack of registers available for caching values.

The question is whether there are higher run time costs associated with an evaluation stack as opposed to general purpose registers.

Properly answering that question is difficult since the ideal approach would be to implement a second, register oriented architecture and to analyze the difference in performance between the two architectures. Those instructions that simply manipulate the stack (e.g., Recover) can be identified as stack oriented overhead that would not occur in a register oriented architecture. The compiler and VlsiCheck executed 3.7% and 6% of their instructions as stack manipulation instructions. This is a minimal cost for a simple stack architecture. The actual cost for Mesa is greater since the number of extra memory loads it performs to keep information on the stack is not available. Naturally, the code compression proffered by the stack oriented Mesa byte codes reduces other costs in the system: a lower bandwidth is required between the memory and the processor for instruction fetches, the compiled code can fit in a machine with less memory, and a machine with a virtual memory will experience fewer code faults.

Emulating 32-bit Data Paths with a 16-bit Processor. The Mesa processor has evolved in the context of 16-bit hardware and much of the instruction set reflects this. Mesa assumes a 16-bit word, and the only way to get 32-bit values is to instruct the compiler that a variable is of type "LONG ...". The default status of 16-bit words becomes apparent when we examine the 32-bit oriented instructions executed by VlsiCheck, which extensively uses 32-bit values, and the compiler, which uses few of them.

Only 20% of VlsiCheck's instructions were 32-bit oriented. Fewer than 1.69% of the instructions were 32-bit arithmetic operations. The VlsiCheck program makes extensive use of 32-bit pointers and arithmetic values: about 15.2% of the instructions were 32-bit loads, stores, or arithmetic computations. An additional 2.7% perform NIL checking on long pointers. The 32-bit instructions must take at least twice as long to execute on a machine with 16-bit data paths. Even though the work of the program uses large values, the majority of instructions are of the "short" kind.

Since so many of the 32-bit instructions reference memory, a useful optimization would provide a pipelined, 32-bit memory reference (over 16-bit data paths) to minimize the overhead of acquiring the second 16 bits.

Only 1.2% of the compiler's instructions deal with 32-bit loads and stores, and a compiler optimization that causes it to generate code for double-word quantities when separate values are consecutive in memory, may account for most of those memory operations.

Instruction Length vs. Frequency of Execution. The smaller the average instruction, the more instructions the processor can acquire in the same unit of time. The preponderance of one-byte instructions reduces the demand for memory bandwith to fetch instructions.

By using the frequency statistics associated with control transfers, branches, and instruction length, we can try to analyze the requirements for an instruction fetch unit. Assume that half the conditional branches cause the processor to change the PC. Then the compiler and VlsiCheck cause the processor to change the PC during 14.2% and 11.3% of the instructions, respectively. This means the processor changes the PC about once every seven or eight instructions. An average instruction length of 1.5 bytes means that an instruction buffer of 16 bytes would accommodate 10 instructions on the average.

If there is a uniform distribution of instruction sequences between branches and procedure calls, there is little reason to have an instruction buffer much bigger than 16 bytes unless the hardware is prepared to handle branches in a sophisticated fashion. Two possible uses for extra instruction buffering are an associative instruction cache or special hardware to follow both sides of a conditional branch [12]. This course introduces much undesirable complication. Since the processor will change the PC every seven or eight instructions, any latency associated with the instruction fetch hardware should be kept low.

Successive Memory Loads: Advantage for the Dorado. The Dorado's processor allows the microprogrammer to pipeline the execution of Mesa instructions [5,13]. In particular, the memory fetch started by one microinstruction may be completed by the next microinstruction. This means that load instructions may require only one microinstruction (start the fetch) rather than two (start the fetch in one cycle and acquire the data the next cycle). The frequency of load-load instructions determines the advantage of this arrangement which costs extra microcode space and makes the microcode more complex. Successive load instructions are 8.4% of VlsiCheck's instructions and 12.8% of the compiler's, and such instruction pairs can exploit the microcode pipelining feature of the Dorado.

This approach is also exploited to save a cycle when dealing with the evaluation stack. Instructions may leave the value of the top of the stack in a special register and leave the stack pointing at the value underneath the top of the stack. This way the ALU Ops, and other instructions that deal with two items on the stack, need not pop one item off the stack to perform an ALU operation on it. (The alternative to this arrangement is to implement a stack with two output ports.) Unfortunately, it is difficult to infer the advantage of this arrangement from these statistics. An unsophisticated inspection of the data indicates that only .35% of the compiler's pairs and 1.6% of VlsiCheck's benefit. The actual percentage is probably much greater; unfortunately, whether an instruction pair benefits from this approach requires a detailed, instruction-implementation dependent analysis that was not performed.

9. Conclusions and Summary

This paper has described and discussed the patterns of instructions executed by two different Mesa programs.

The statistics show that the compiler's optimizations for code compaction are a good predictor for the frequency with which variables are referenced dynamically. In particular, the first few Locals and Globals account for most references in those frames. This result suggests that a special cache for the first few Local and Global variables would effectively reduce the number of memory references required by the instruction set.

The unique cooperation between the compiler and the Mesa processor effectively reduces the load on the processor in two ways: Instructions are shorter and operands' effective addresses are easily computed. It might be possible to exploit this fact with hardware that prefetches operands for the "zero offset" instructions, although that might not be worthwhile if the locals-cache described above were implemented.

Fewer than 20% of VlsiCheck's instructions were 32-bit instructions. We see that a program that extensively exploits 32-bit values and "full length" pointers (processor implementation dependent size) pays a corresponding penalty for 16-bit data paths: 20% of the instructions require an extra memory reference. This presumes, of course, that the compiler can recognize the variables that need be only 16-bits wide and properly optimize the code that manipulates them. The exact time penalty will be a function of the processor implementation.

References

1. Geschke, C.M. *et al.* Early experience with Mesa. *Comm ACM*, **20**, 8, Aug. 1977, pp. 540-552.

2. Mitchell, J.G. *et al. Mesa Language Manual,* Technical Report CSL-79-3, Xerox Palo Alto Research Centers, 1979.

3. Lampson, B.W., Mitchell, J., and Satterthwaite, E. On the transfer of control between contexts. *Lecture Notes in Computer Science 19,* 1974.

4. Thacker, C.P. *et al.* Alto: A personal computer. In *Computer Structures: Readings and Examples,* 2nd edition, Sieworek, Bell and Newell, eds., McGraw-Hill, 1981. Also in Technical Report CSL-79-11, Xerox Palo Alto Research Centers, Aug. 1979.

5. Lampson, B.W. *et al.* The Dorado: Three Reports, Technical Report CSL-81-1, Xerox Palo Research Centers, Jan. 1981.

7. Lampson, B.W. and Redell, D.D. Experience with processes and monitors in Mesa. *Comm. ACM 23,* 2 Feb. 1980, pp. 105-117.

8. Johnsson, R. and Wick, J. An Overview of the Mesa Processor Architecture, *Symposium on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, CA, 1982.

9. Sweet, R. and Sandman, J. Empirical Analysis for Mesa Instruction Set Design, *Symposium on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, CA, 1982.

10. Deutsch, L.P. A Lisp Machine with very compact programs, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence,* Stanford, CA, 1973, pp. 687-703.

11. Tannenbaum, A.S. Implications of structured programming for machine architecture, *Comm ACM 21*, 3 Mar. 1978, pp. 237-246.

12. Anderson, D.W. *et al.* The System/360 Model 91: Machine philosophy and instruction handling. *IBM J. R&D*, 11, 8, Jan. 1967, pp. 8-24.

13. Lampson, B.W., McDaniel, G.A., and Ornstein, S.M. An Instruction Fetch Unit for a High Performance Personal Computer. The Dorado: Three Reports, Technical Report CSL-81-1, Xerox Palo Research Centers, Jan. 1981. Revised version to appear in *IEEE Transactions on Computers.*

14. Lampson, B.W. Fast Procedure Call. *Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, 1982.

15. Connors, W.D. *et al.* The IBM 3033: An inside look. *Datamation*, May 1979, pp. 198-218.

Acknowledgments. Doug Clark, Neil Wilhelm, and Jim Gray deserve special thanks for their efforts with this paper, along with many of my colleagues in CSL.

Appendix: Instruction Descriptions

LLi, LGi, SLi, SGi
: Load or Store from the Local or Global Frame the i^{th} variable

LLB, LLDB, SLB, SLDB
: Load or Store from the Local or Global Frame given a byte offset. "D" indicates a double-word quantity

RF
: Read a bit field from a 16-bit value

Ri
: Read the i^{th} word from the pointer on the top of the stack

RXLP, RILP
: Read a value, indexed or indirect with post indexing

JZNEB, JZEQB, JEQB
: Conditional branches with a byte offset for the PC

LIW, LIB, Li
: Load immediate values (word, byte, small constant)

RECOVER
: Recover the previous top of stack by incrementing the stack pointer without modifying the contents of the stack

MUL, ADD, SUB, INC
: Arithmetic operations

BNDCK, NILCK, NILCKL
: Boundary and pointer check instructions